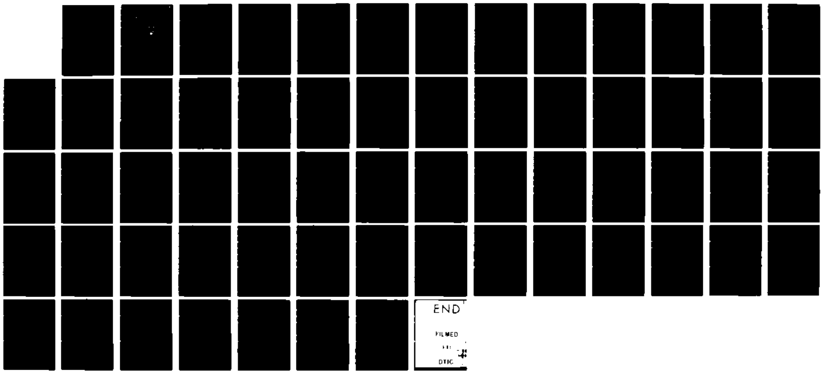
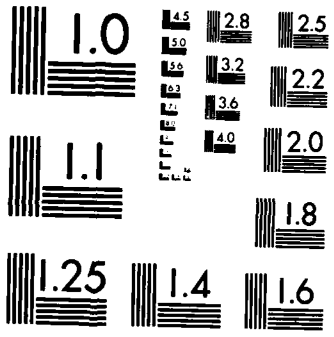
AD-A125 690   A VIEW OF OBJECT-ORIENTED PROGRAMMING(U) NAVAL   1/1
POSTGRADUATE SCHOOL MONTEREY CA B J MACLENNAN FEB 83
NPS52-83-001 N00014-83-WR-20162

UNCLASSIFIED   F/G 9/2   NL




1.0
4.5
5.0
5.6
6.3
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

2

AD A125690

NPS52-83-001

NAVAL POSTGRADUATE SCHOOL

Monterey, California

A VIEW OF OBJECT-ORIENTED PROGRAMMING

Bruce J. MacLennan

February 1983

Prepared for:

Naval Postgraduate School
Monterey, CA 93940

83 03 14 166

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral J. J. Ekelund
Superintendent

D. A. Schrady
Provost

The work reported herein was supported in part by the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of Naval Research.

This report was prepared by:

[illegible]

BRUCE J. MACLENNAN
Assistant Professor of
Computer Science

Reviewed by:

[illegible]

DAVID K. HSIAO, Chairman
Department of Computer Science

Released by:

William M. Tolles

WILLIAM M. TOLLES
Dean of Research

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-83-001	2. GOVT ACCESSION NO. AD-A125 698	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A View of Object-Oriented Programming		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Bruce J. MacLennan		8. CONTRACT OR GRANT NUMBER(s) N00014-83-WR-20162
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61152N; RR000-01--10 N0001482WR20043
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		12. REPORT DATE February 1983
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Object-oriented Programming, Value-oriented Programming, Applicative Programming, Functional Programming, Production Systems, Data-flow Languages, Information Hiding, Capabilities, Concurrency, Synchronization.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The differences between values and objects and, hence, between value-oriented and object-oriented languages are clarified. It is argued that languages should have both values and objects; the proper role of each is described. A proposal is presented for a true object oriented programming system. This includes both an informal description of object-oriented programming constructs and a formal semantics for these constructs. Non-determinancy, synchronization and recovery from failures are briefly discussed.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S N 0102- LF- 014- 6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

A VIEW OF OBJECT-ORIENTED PROGRAMMING

B. J. MacLennan

Computer Science Department

Naval Postgraduate School

Monterey, CA 93940

CONTENTS

A VIEW OF OBJECT-ORIENTED PROGRAMMING

B. J. MacLennan

Computer Science Department

Naval Postgraduate School

Monterey, CA 93940

Abstract.

Applicative programming languages and languages for data-flow machines are often described as value-oriented languages. Other languages, such a Smalltalk, are described as object-oriented. LISP has been described as both value-oriented and object-oriented. What exactly do these terms mean?

This paper attempts to identify and clarify the differences between values and objects and, hence, between value-oriented and object-oriented languages. The paper then turns to the question of whether objects should be included in applicative languages and the role they can fill in those languages. The remainder of the paper is a proposal for one approach to a true object-oriented programming. This includes both an informal description of object-oriented programming constructs and a formal semantics for these constructs. Nondeterminacy, synchronization and recovery from failures are briefly discussed.

1. Introduction

1.1 What are Values?

Value-oriented programming can be described as programming without the use of variables, side-effects, or an updatable memory. The basic idea is that the value of an expression depends only on its operands, and that the only effect of executing an expression is the value it returns. Thus the operators in an expression are mathematical functions.

Almost every programming language has a value-oriented subset: its arithmetical expressions. The value of an expression such as (3 + 2) depends only on its input operands (3 and 2, in this case), and the only effect of executing the expression is the value, 5, returned.

One of the desirable characteristics of values is their predictability. We can tell the interfaces of an arithmetic expression by simple inspection. Since all of the functions are side-effect-free, all of the inputs and outputs of each functions are manifest. This simplifies manipulation of value-oriented expressions, and simplifies proving properties about them.

What is it about values that give them these characteristics? We have noted that the most value-oriented parts of most programming languages are there most mathematical parts, so it is no surprise to find that mathematics is that discipline that most consistently deals with values. Hence, we can discover many of the characteristics of values by studying mathematical entities.

What are these characteristics? The fundamental property of mathematical entities is that they are abstractions (concepts, universals). For example, the number two is an abstraction that subsumes as its particulars all of the various pairs, whether they exist in reality or our imagination, or in the present, or the past or future. Because mathematical entities are abstractions, they do not change. Particular pairs may come into or go out of existence, but the abstraction two remains. Also, the number of marbles in a bag may be subsumed by the abstraction two at one time, and the abstraction three at a later time, but neither abstraction has been altered. Similarly, it is not meaningful to speak of the creation or destruction of a value; mathematical equations such as 5 = 2+3 describe timeless relationships among values, not descriptions of their creation, modification, or destruction. In this sense values are atemporal; that is, the concept time does not apply to them. It is just as meaningless to apply time concepts to a number as it is to apply color concepts.

Another characteristic of values is that they are universal; that is, they are not particular. This is because an abstraction is coextensive with the particulars it subsumes. For example, the abstraction two is coextensive with all particular pairs, and conversely, anything coextensive with all particular pairs is the abstraction two. The result of this is that it makes no sense to talk of "this number two" or "that number two" or to ask, "How many number two's are there?" These are concepts that apply to particular things, and values are universals, not particulars.

The characteristics of values can be summarized:

- values are atemporal;
- values are immutable;
- values are neither created nor destroyed;
- values are universal.

It can be seen now that applicative programming is essentially value-oriented programming. In a purely applicative language, there is no assignment operation and no idea of state. Functions compute values solely on the basis of their inputs, and have no side-effects. This is of course why applicative programming is so mathematical; it is mathematics.

Consider an applicative subset of LISP (i.e., without RPLACA and RPLACD, and with EQ restricted to atoms). In this language lists can be treated like mathematical entities (specifically, tuples) and the list processing operations can be treated like mathematical functions on these entities. Lists may be computed, but they are not modified, created, destroyed, copied, or shared. Again, applicative programming (value-oriented programming) is essentially mathematics.

1.2 What are Objects?

There is probably more confusion about the nature of object-oriented programming than about the nature of value-oriented programming (see, for example, [Lomet76], [Lomet80], [Robson81], [Rentsch82] and [MacLennan82]). In this section we identify some of the characteristics of objects and object-oriented programming.

In object-oriented programming systems such as Smalltalk, all computation is viewed as simulation. Here, programming language objects correspond to real-world objects, and manipulations of real-world objects are simulated by sending messages to the programming language objects. In Smalltalk, programming language objects are grouped into _classes_ (i.e., abstractions) of similarly behaving objects.

What is an object? We can get a clue to its nature from the common use of the term. In common usage an object is a material entity existing in space or time. Let us consider what it means for a programming language object to exist _in time_. This means that it persists as an identifiable entity through some period of time, which in turn means that it can come into existence at some point in time (i.e., be created) and that it can go out of existence at some point in time (i.e., be destroyed). This is a common characteristic of objects in computers; for example, objects in Smalltalk and Simula are created by explicit request (the _new_ operation), and are destroyed by a garbage-collector when they are no longer accessible.

Just as real objects exist _in space_, programming language objects may enter into a number of relationships, spatial and otherwise, with other objects. For example, one object may be _part_ of another object, or (in an operating system) the _owner_ of another object. In a system like Smalltalk the attributes of an object are represented by instance variables, whose values are themselves objects. In data bases intended to represent real-world knowledge, nodes representing objects are connected by labeled arcs representing relationships among the objects.

Real-world objects not only _exist_ in time, they also change _through_ time. That is, various factors can alter an object's relationships with other objects (such as position) and an object's other attributes (such as size). This is in marked contrast to values, which, as we have seen, are immutable. Another way to state this is that at any point in time an object has a _state_, which is the sum-total of its relationship with all other objects in the system. Various laws then determine how the state of an object can change in time. Of course, the ultimate change in state that any object can undergo is its creation or destruction.

In systems like Smalltalk the instance variables determine the state of an object and the methods defined in the object's class determine the object's behavior in time.

The mutability of objects leads to another of their important characteristics: the notion of sharing. We have said that values are universal, i.e., that the concept of _instance_ does not apply to them. This is not the case for objects. Since only a finite number of properties are representable on a finite computer system, it is quite possible that two programming language objects have all the same properties, yet represent distinct real-world objects. It is of course quite common in object-oriented programs to have two different objects that at some point in time have the same attributes. Thus, objects have an _identity_ independent of their state, which makes them _particular_ rather than _universal_. If two entities _share_ access to the _same_ object, then if one entity changes the state of the object, the change will be visible to the other entity. Conversely, if the two entities each have access to _distinct_ objects (that might have the same state) then a change to the state of one will not directly cause a change to the state of the other. The shared/non-shared distinction is of course familiar from object-oriented programming systems.

The characteristics of objects can be summarized:

- objects are temporal; they exist in time;
- objects are mutable, and have a state;
- objects can be created and destroyed;
- objects are particular, and can be shared.

1.3 _Objects in Applicative Languages._

It would seem that objects should be excluded from applicative programming systems. They have many of the undesirable characteristics of imperative languages, such as a changeable state, a strong dependence on the time sequence of events, and complications arising from the notion of sharing. In this section we will argue that this is not the case, that objects have a role in applicative languages.

The reason for this is simple: the purpose of a program is often, directly or indirectly, the modeling of some aspect of the real-world. Further, the aspects that we are interested in modeling often involve the changing relationships among real-world objects. The obvious approach is to use programming language objects to model the real-world objects.

Given this observation it is not surprising that the concepts of object-oriented programming first arose in connection with simulation languages, in particular Simula [Dahl70]. Smalltalk, one of the most widely known object-oriented languages, is based on Simula and takes the view that all programming in simulation [Kay77]. For these reasons many basic ideas of simulation (see for example [Pritsker79], Chapter 3 and [Maisel72], Chapter 1) are fundamental to our notion of objects.

It will be objected that applicative languages do not need objects to model the changing state of real-world objects. The technique is familiar from the use of denotational semantics to describe imperative languages. The total state of the system is described by some <u>value</u>, such as a sequence or a function. This value is the input to an applicative state-transition function that computes the value representing the new state.

Although this works, it is not very satisfactory. Functions tend to accumulate large numbers of arguments, or highly structured composite arguments, that represent the state. This defeats the goals of applicative programming by destroying the clarity and mathematical tractability of programs. The problem becomes acute in operating-systems, real-time systems, graphics software, data-base systems, and of course in explicit simulations. For this reason the remainder of this paper will discuss how objects can be best reconciled with applicative languages.

We must first note an obvious point: the entire state of the universe cannot be simulated inside a computer. Therefore, it becomes necessary to select the subpart of the universe relevant to the problem, and the appropriate level of abstraction for the simulation.

The result is that the state of the simulation is represented by a finite number of objects connected by a finite number of relationships. As the simulation progresses the relationships among the objects may change and objects may be created or destroyed.

The relations and objects described above can be termed corresponding objects and relations because they correspond to the objects and relations in the real-world that they model. The system may also contain non-corresponding objects and relations.

These do not necessarily correspond to any real-world objects or relations, but are used to implement the corresponding objects and relations. Thus, the distinction between corresponding and non-corresponding objects and relations is analogous to the distinction in object-oriented programming languages between the publicly visible objects and their abstract properties on one hand, and the private objects and attributes used to implement the public ones on the other.

The non-corresponding objects are analogous to theoretical entities in a scientific theory: we cannot in general infer from them the existence of corresponding real-world objects. Thus, like the implementation details of a Smalltalk or Simula object, it is usually desirable if the non-corresponding objects and relations be hidden. This can be accomplished by the proper control of name contexts. That is, an entity can only gain access to an object or relation if that object or relation has a name in a context accessible to that entity. This is analogous to having a capability for an object in an accessible capability list. Generally, only the implementor of a class of objects will have access to a context naming the non-corresponding objects and relations used to implement that class of objects. Thus, information hiding is easily accomplished.

The objects and relations that are intended to be corresponding can be made public by giving them a name in a more widely accessible context. Of course, the changeable name contexts that are used to accomplish this control are themselves objects.

How do the relations among objects come to be changed? In the real-world such changes are expressed by _causal laws_, which state how certain relations holding among objects _cause_ these relations to change in time. These laws are usually expressed as a conditional statement about some class of objects. For example, "an electron in conditions C will act in manner A."

The same approach can be used for controlling the changing relations among objects in a programming system. This is similar to what is done in object-oriented languages like Smalltalk: classes of objects are defined which behave in the same way in various message-receiving situations. Thus, the _methods_ of Smalltalk can be thought of as causal laws.

How do these ideas relate to applicative programming languages? Note that values enter into the universe of objects in several places. First, certain attributes of objects (e.g., weight) and relations among objects (e.g., distance) will be values. Second, the relations that hold among objects at any given time are themselves values, since relations are mathematical abstractions.

At each instant of time the causal laws must determine new relations to be associated with the relation-names at the next instant of time. These changes are expressed as _transactions_ that add tuples to, or delete tuples from, the relations. Applicative programs can be used to determine the values to be associated with the value-bearing attributes of an object. Thus, value-oriented and object-oriented programming techniques can be used together in a way that exercises the advantages of each. Values and pure functions are used to model abstractions and their relationships, while objects and causal laws are used to model real-world objects and their behavior.

2. Intension versus Extension

It is necessary to distinguish between the intension and extension of the relations in the computer that are used to model corresponding external relationships and properties. Two relations have the same intension if they are intended to model the same external relationships or properties. Two relations have the same extension (at a given point in time) if they apply to the same objects (or tuples of objects and values). Notice that two relations may be extensionally the same even though they are intensionally different. For example, at a given point in time, the same three objects may satisfy both the relations "blue" and "round," but this does not imply that "blue" and "round" model the same property. That is, "blue" and "round" have the same extension but different intensions. Also, notice that one of the objects might at the next instant become "non-blue" while remaining "round." Thus, although the intension of relations remains fixed their extension can vary in time. From time to time, relations with different intension can coincidently have the same extension.

The same distinctions apply to objects. The intension of a computer object is the real-world object it is intended to model; the extension of a computer object is the set of relations to which it belongs. Thus objects with different intensions can coincidently have the same extension.

These notions of intension and extension might seem to conflict with Liebnitz' doctrine of the identity of indiscernibles, which says that two things that are alike in every way are identical. Symbolically,

$$x = y \Leftrightarrow \forall P\{P(x) \Leftrightarrow P(y)\}$$

In the real world two things which agree in every property and relationship are the same thing. In a computer system, however, it is not possible to model every property and relationship; it is necessary to select a finite number of these that are relevant to the problem at hand. Therefore we can have two computer objects that are intended to model distinct real-world objects, but happen to agree in all the modeled properties and relationships. That is, the two computer objects have different intensions but the same extensions. These ideas are developed further in [MacLennen73] and [MacLennan75], Chapter 3, which present a mathematical theory of intensional relations and sets.

How can a programming system distinguish intensionally distinct relations that happen to have the same extension? Although there are a number of solutions to this problem, the simplest is to associate a unique ID with each intensionally distinct object or relation. Then, two objects or relations are distinct if and only if they have distinct IDs. At each point in time each relation ID is associated with a relation (i.e., a set of tuples) that defines the relation's extension.

Notice that these IDs are internal identifiers analogous to references or capabilities; they have no connection with any names that programmers might use to refer to these objects or relations. In fact, we will see later (Section 7) that programmers manipulate objects and relations by giving names to their IDs.

3. Condition-Action Rules

We describe the behavior of objects by using causal laws of a special form, condition-action rules. A condition-action rule says that if some objects are in a certain situation they will act in a certain way. That is, if certain relations do or do not hold for a particular objects, then those objects will establish or disestablish certain other relations. We write these rules in the form

$$\langle\text{cause}\rangle \Rightarrow \langle\text{effect}\rangle$$

where <cause> defines the conditions under which the objects act, and <effect> defines the actions that they take under those conditions. These rules are very similar to production rules; we explore the differences later (Section 9). Next we will discuss the effect part of rules; the cause part is discussed later.

We use a prefix notation for expressing actions; that is, we use

P (x, y, z, ...)

to mean that the tuple <x, y, z, ...> is to be added to the relation P. Similarly,

~P (x, y, z, ...)

means that the tuple <x, y, z, ...> is to be deleted from the relation P. In the above x, y, z, ... represent either simple variables or <u>constructor</u> expressions, that is, expressions that construct compound structures from other simple or compound structures. For example, the constructor expression cons(x,y) constructs the compound structure <cons,x,y>.

Since a single rule can cause a number of actions to occur, the <effect> part of a rule allows the specification of a set of additions or deletions. For example, the rule

... $\Rightarrow$ Contents(s,y), Receive(x)

causes the tuple <s,y> to be added to the Contents relation and the tuple <x> to be added to the Receive relation. As we said, the tuples can contain the results of constructor expression evaluation. For example, the rule

... $\Rightarrow$ Contents(s, cons(x,y)), a(s)

causes the tuple <s, <cons,x,y>> to be added to the Contents relation, where <cons,x,y> is the result of evaluating cons(x,y) in the current context. The names "s," "Contents," "cons" etc. must be bound in the current context, which is discussed later.

Conditions are specified by pattern matching. In a simple case such as

Push(s,x,a), Contents(s,y) $\Rightarrow$ ...

we are testing if there is a tuple <s,x,a> in the Push relation and a tuple <s,y> in the Contents relation. The meaning of this condition depends on the bindings of the names that occur in it, "s," "x," "a" etc. in this case. Any name that is bound to a value or object ID will match that value or object ID; any name that is unbound will match any value or object ID and bind the name to that value or object ID. In the above example suppose that all the lowercase names are initially unbound. Then Push(s,x,a) will match any triple in the Push relation and bind its components to "s," "x" and "a." Next, Contents(s,y) will match any pair in the Contents relation whose first element is the "s" matched in the first condition. If this match succeeds then "y" will be bound to the second element of the pair. Thus we have tested whether any object is both a first member of Push and a first member of Contents.

The above conditions have a side effect of binding the names "s," "x," "y" and "a" to the components of the tuples that satisfy the conditions. These bindings remain in effect during execution of the effect part of the same rule; they are then discarded. This allows us to perform actions on the values and objects satisfying the condition.

In the above example we tested for the presence of a condition; we can also test for the absence of a condition. Had we written

```
Push(s,x,a), ~Contents(s,y)  ⇒  ...
```

we would have executed the action part if there were not a pair in Contents whose first element was the same as the first element of any triple in Push.

With this explanation the meaning of a complete rule can be understood. Consider:

```
Push(s,x,a), Contents(s,y)  ⇒  ~Push(s,x,a), ~Contents(s,y),
  Contents(s, cons(x,y)), a(s)
```

This means that if the Push relation holds for some objects s, x and a, and if s holds the Contents relation to some object y, then perform the following actions:

1. Disestablish the Push relation between s, x and a.

2. Disestablish the Contents relation between s and y.

3. Establish the Contents relation between s and cons(x,y).

4. Establish the a relation on s.

We can see that s is a stack. The meaning of the Push relation is that some agent is attempting to push the object x on stack s and is expecting an acknowledgement in a. The stack accomplishes this by altering its Contents relation. It acknowledges its action by placing itself in the Receive relation. An alternative interpretation is that some agent sends the message Push(s,x,a) to the stack. The stack responds by altering its Contents and sending the stack back through a. Typically Push would be a corresponding (public) relation and Contents would be a non-corresponding (private) relation.

The above rule demonstrates a common situation: the conditions that hold in the cause are disestablished in the effect. For this reason we define an abbreviation: any rule with a condition of the form *<name><tuple> can be replaced by a rule in which this condition is replaced by <name><tuple> and an additional action ~<name><tuple> is added. Thus, any tuple successfully found by a match of this kind is deleted from the relations in which it was found before the action part of the rule is executed. This automatic updating is convenient since it is the usual case. Using it the Push rule can be written:

*Push(s,x,a), *Contents(s,y) $\Rightarrow$ Contents(s, cons(x,y)), a(s)

These rules are similar to operation nodes in a data-flow language: when tuples of a certain form arrive in the input relations Push and Contents, the rule "fires" by removing the tuples from the input relations and putting other tuples in the output relations Push and a.

Popping from a stack illustrates the use of more complex patterns. Consider

*Pop(s,a), *Contents(s, cons(x,y)) $\Rightarrow$ Contents(s,y), a(x)

The first clause in the condition searches for any pair <s,a> in the Pop relation. The interpretation is that some agent is asking a stack for its top element and wishes it to be returned in a. The second clause in the condition searches the relation Contents for any pair whose first element is that same stack and whose second element is something that matches the pattern cons(x,y). If we assume that "cons" is already bound, then the pattern cons(x,y) matches any triple whose first element is the value of "cons." The remaining two elements of this triple are bound to "x" and "y." The effect of this clause is to take the triple that is the stack's contents and decompose it into its components. Thus this rule reverses the effect of the Push rule, as expected. Notice that rules are symmetric: the same relation and constructor expressions can be used on either side.

4. Indivisibility of Rules

So far we have discussed the form of rules but not the execution cycle of the abstract machine on which they execute. Although this will be defined precisely in Section 8, Formal Semantics, we now address the issue informally. The basic rule of execution is that on each cycle of the abstract machine one rule is applied. That is, out of all the rules whose conditions are satisfied, we arbitrarily select one and execute its actions. The effect of this is that rules are executed indivisibly.

The indivisibility of rules is important in a number of applications that require the synchronization of concurrent activities. For example, if there is exactly one x such that MutEx(x), then we can ensure the mutual exclusion of two processes by:

```
*MutEx(x)  ⇒  ... begin process A ...
... finish process A ...  ⇒  Mutex(x).
*Mutex(x)  ⇒  ... begin process B ...
... finish process B ...  ⇒  MutEx(x).
```

Since a rule is executed to completion in one cycle, either process A or process B is *guaranteed to get exclusive access*. In the next section we apply this idea to a simple producer-consumer synchronization problem.

5. Examples

In this section we present several simple examples of sets of rules. First, extending the examples of Section 3, we have these rules for stacks:

```
*NewStack(a), *Avail(s)  ⇒  Contents(s,nil), a(s).
*Push(s,x,a), *Contents(s,y)  ⇒  Contents(s,cons(x,y)), a(s).
*Pop(s,a), *Contents(s,cons(x,y))  ⇒  Contents(s,y), a(x).
*Destroy(s,a), *Contents(s,x)  ⇒  a(x).
```

Notice that we have added rules for both creating and destroying stacks. The creation rule fetches an unused object from the relation Avail to make into a stack. Typically NewStack, Push, Pop and Destroy would be public names and Contents would be private to the implementor.

We could have added to these rules a relation Stack(s) that asserts that s is a stack. This relation would effectively define the type of s. It is not necessary to define this relation since we can classify anything in the first position of a tuple in Contents as a stack. The integrity of the type is preserved by keeping the Contents relation private to the implementor.

To see how concurrent processes can use this model for synchronization, consider a simple producer sending messages to a consumer through an unbounded buffer. An agent puts a message m in the buffer by sending Produce(m,p). An agent consumes a message by sending Consume(c) and receiving the next message m by *c(m). This is expressed by the following rules:

```
*Initialize(a)  ⇒  Pindex(0), Cindex(0), a(0).
*Produce(x,a), *Pindex(k)  ⇒  Buffer(k,x), Pindex(Suc(k)), a(k).
*Consume(a), *Cindex(k), *Buffer(k,x)  ⇒  Cindex(Suc(k)), a(x).
  where Suc(x) denotes the successor of x.
```

Notice that the indivisibility of rules ensures that simultaneous Produce messages will get distinct Pindexes and that simultaneous Consume requests will get consecutive buffer elements.

6. Notational Extensions

The syntax for condition-action rules described previously is termed the canonical form for rules. In this section we make several notational extensions to the canonical form to simplify expressing rules.

6.1 Compound Rules

Suppose we wished to write rules that perform one action if P contains a pair of the form <x,x> and a different action if it contains any other pair <x,y>. The following to rules will not accomplish this, since a pair <a,a> will match the pattern P(x,y):

$$P(x,x) \Rightarrow \text{Action 1.}$$
$$P(x,y) \Rightarrow \text{Action 2.}$$

What we would like to say is: first try the pattern P(x,x) and only if this fails try P(x,y). This idea can be expressed by using a negative condition:

$$P(x,x) \Rightarrow \text{Action 1.}$$
$$\sim P(x,x),\ P(x,y) \Rightarrow \text{Action 2.}$$

We allow the following notational abbreviation for this common case:

$$P(x,x) \Rightarrow \text{Action 1}$$
$$\text{else } P(x,y) \Rightarrow \text{Action 2.}$$

This extends in the obvious way to more that one else-arm and to more than one condition in the cause parts.

6.2 Sequential Blocks

Often we want the mechanism of an object to move sequentially through two or more states. This can be programmed explicitly by using a relation, say $\emptyset$, to represent the state of the object. For example,

$$\begin{array}{rcl} C_0 & \Rightarrow & E_0,\ \emptyset(1,\ \overline{v}). \\ *\emptyset(1,\ \overline{v}),\ C_1 & \Rightarrow & E_1,\ \emptyset(2,\ \overline{v}). \\ \cdots\cdot & & \\ *\emptyset(n,\ \overline{v}),\ C_n & \Rightarrow & E_n. \end{array}$$

where $\overline{v}$ represents all the unbound variables of C_0. This allows a rule to fire only if the object is in the proper state. We allow a group of rules of the above form to be written as a <u>sequential block</u>:

$$C_0 \Rightarrow \{E_0;\ C_1 \Rightarrow E_1;\ \ldots\ ;\ C_n \Rightarrow E_n\}$$

The semicolons are suggestive of the sequential execution of statements in conventional programming languages. Notice that the variables in C_0 essentially become global variables of the entire block.

Sometimes rules in sequential blocks have empty cause parts, for example "$\Rightarrow E$," since the rule is to be applied unconditionally when the object is in the proper state. In these cases we allow the arrow to be dropped: "E."

6.3 Procedure Calls

The communication mechanisms we have described are asynchronous, that is, a message is sent by an action such as Lookup(d,n,a) (a request to look up name n in directory d and return the result in a) and a result is received by a condition such as *a(x). Any amount of processing might be done by the sender between the Lookup and the reply.

In many situations the sender cannot go on; it must wait for a reply. For example,

```
*R(n,a)  ⇒  Lookup( Public, n, Receive), ∑(a).
*Receive(s), *∑(a)  ⇒  Pop(s,a).
```

where we assume the only purpose of ∑ is to convey a from the first rule to the second. In these cases we are doing synchronous communication. Since synchronous communication is so common, we allow the above example to be written:

```
*R(n,a)  ⇒  Pop( Lookup[ Public, n], a).
```

The square brackets indicate that Receive is to be passed as the last argument and that we are to wait for a reply (through the private Receive relation) before continuing. A relation name followed by an argument list in square brackets is termed a procedure call or, more briefly, a call.

Next we consider a slightly more complicated example. Suppose that we have two nested calls; what would this mean?

$$*R(n,a) \Rightarrow a(\ Pop[\ Lookup[\ Public,\ n]\ ]\ ).$$

If we reduce this in the same way as the previous example we get the three rules:

$*R(n,a) \Rightarrow Lookup(\ Public,\ n,\ Receive),\ \Sigma(n,a).$

$*Receive(s),\ *\Sigma(n,a) \Rightarrow Pop(\ s,\ Receive),\ \Sigma(n,a).$

$*Receive(x),\ *\Sigma(n,a) \Rightarrow a(x).$

A problem is apparent: The last two rules have essentially the same left-hand sides; this means that either of these rules could accept the result returned by Lookup, which is incorrect. To ensure the proper synchronous communication these rules must be more tightly bound. Since a particular <u>instantiation</u> of a rule is uniquely determined by the rule and the bindings performed by the cause part, we can tag each communication with this information. Therefore, to reduce the rule

$$*R(n,a) \Rightarrow P(\ Pop[\ Lookup[\ Public,\ n]\ ]\ ).$$

to the canonical form, we must create new relations Σ, p and σ, and replace the rule by these three:

$*R(n,a) \Rightarrow Lookup(\ Public,\ n,\ p),\ \Sigma(n,a).$

$\Sigma(n,a),\ *p(s) \Rightarrow Pop(s,\sigma).$

$*\Sigma(n,a),\ *\sigma(x) \Rightarrow a(x).$

The separate private relations p and σ are used to distinguish different acts of communication in the original rule.

This process can be generalized in a straight-forward way to handle effects that contain any number of synchronous calls. To show this we present an algorithm that reduces a rule containing any number of synchronous calls to a set of rules in the canonical form. We suggest that this description be skipped on a first reading.

First, rearrange the rule so that it has the form

$$C(V) \Rightarrow A_1, \ldots, A_m, B_1, \ldots, B_n \qquad (1)$$

where the A_i are the actions containing calls and the B_i are actions not containing calls. C(V) represents a cause part containing the free variables

$$V = v_1, \ldots, v_f$$

Number all the calls in the rule (1) from 1 to N. Invent new private names $r_1, \ldots, r_N$ and $\rho_1, \ldots, \rho_N$. The ρ_i will be the relations used to receive the values from the calls; the r_i will be bound to the returned values. Create new relations Π, Σ, Δ, and $\phi_1, \ldots, \phi_N$. The relations ϕ_i will be used to receive the reply relations to be bound to ρ_i.

Replace rule (1) by the rules:

$$C(V) \Rightarrow \Pi(V),\ B_1,\ \ldots,\ B_n,\ NewRel(\emptyset_1),\ \ldots,\ NewRel(\emptyset_N). \quad (2)$$

$$*\Pi(V),\ \emptyset_1(p_1),\ \ldots,\ \emptyset_N(p_N) \Rightarrow \Sigma(U). \quad (3)$$

$$\begin{array}{ll} \Sigma(U) \Rightarrow A_1. & \\ \quad \ldots & (4) \\ \Sigma(U) \Rightarrow A_N. & \end{array}$$

$$*\Sigma(U) \Rightarrow . \quad (5)$$

where $U = V,\ p_1,\ \ldots,\ p_N$. Rule (2) captures the parameters in relation Π, initiates all the actions that do not contain synchronous calls and initiates requests for N new reply relations. (NewRel is a public relation that provides previously unused relation IDs.) Rule (3) receives the N new reply relations and combines them with the parameters into a unique activation record in the Σ relation. (4) denotes a set of rules - one for each action containing synchronous calls. These rules will be processed in later steps of the algorithm to eliminate these calls. Rule (5) will be modified in later steps of the algorithm to perform clean-up functions such as deleting the activation record.

For each rule in the set (4), as long as that rule contains a synchronous call, write the rule in the form

$$\Sigma(U),\ p_1(r_1),\ \ldots,\ p_m(r_m),\ \ldots,\ p_n(r_n) \Rightarrow E(f[X]). \quad (6)$$

where by E(f[X]) we mean an action containing the call f[X], where X is any actual parameter list. The first time this step is performed n will be zero (i.e., there are no "$p_i(r_i)$"). Recall that at the beginning of the algorithm we numbered the calls from 1 to N; suppose that f[X] is the k-th call. Further, suppose that the conditions have been reordered so that r_1, ..., r_m occur in X and r_{m+1}, ..., r_n occur in E(-). Rewrite rule (6) as these two rules:

$$\Sigma(U),\ p_1(r_1),\ \ldots,\ p_m(r_m) \Rightarrow f(X, p_k). \tag{7}$$

$$\Sigma(U),\ p_{m+1}(r_{m+1}),\ \ldots,\ p_n(r_n),\ p_k(r_k) \Rightarrow E(r_k). \tag{8}$$

Rule (7) initiates the execution of f; rule (8) waits for its result and continues the execution of E.

The above process is continued until there are no more calls in the set (4). Call the relations updated in the actions A_1, ..., A_m the <u>final actions</u>. The above reduction process will result in m rules of the form (8), one for each final action:

$$\Sigma(U),\ p_1(r_1),\ \ldots,\ p_n(r_n) \Rightarrow A_i(Y). \tag{9}$$

We replace each of these rules by:

$$\Sigma(U),\ p_1(r_1),\ \ldots,\ p_n(r_n) \Rightarrow A_i(Y),\ \Delta(p_1,\ \ldots,\ p_n). \tag{10}$$

The purpose of the relation Δ is to signal the completion of the final actions. For each such rule (10) created we add the condition $*\Delta(p_1, \ldots, p_n)$ to the cause part of rule (5). When this has been done for all the final actions, the modified rule (5) will have the form:

$$*\Sigma(U),\ *\Delta(\ldots,\ p_i,\ \ldots),\ \ldots,\ *\Delta(\ldots,\ p_j,\ \ldots) \Rightarrow\ . \tag{11}$$

When each of the m final actions has been completed, this rule deletes the activation record in Σ and the completion signals in

△.

When this algorithm has been completed we will have a set of rules with this structure (rule numbers are shown in parentheses):

- Creation of parameter record and reply relations (2)
- Creation of activation record (3)
- Call processing (7, 8)
- Final actions (10)
- Destruction of activation record (11)

6.4 Valueless Procedures

The procedures described above are <u>value</u> <u>returning</u>; they return a value that is used in the expression in which the call occurs. In sequential blocks it is often useful to have <u>value-less</u> procedures, that is, procedures that have an effect but do not return a value. For example, the block

```
{  ... ;
   Push[S,x];
   R(2)  }
```

in which Push is valueless, can be reduced to this block:

```
{  ... ;
   Push(S,x,p);
```

```
    *p(y)  ⇒  R(2)  }
```

In this case *p(y) waits for an acknowledgement and ignores the returned value.

6.5 Applicative Expressions

It is often useful to be able to evaluate applicative (value-oriented) expressions in the effect part of rules. For example, suppose that we represent name directories by association lists. The rules to put names in these directories and to look up their values can be written:

```
*Define(d,n,x,a), *Contents(d,y)
  ⇒  Contents( d, cons( pair(n,x), y)), a(d).
*Lookup(d,n,a), Contents(d,x)  ⇒  a(assoc(n,x)).
```

In the second rule above assoc(n,x) is interpreted as an application of the function assoc to the values n and x. This rule can be reduced to the following, which explicitly calls for the evaluation of the expression:

```
*Lookup(d,n,a), Contents(d,x)  ⇒  a( Eval[ assoc(n,x), Current] ).
```

where assoc(n,x) is now interpreted as a simple data structure constructor and Current is the current environment.

Since a rule must always be executed to completion before another rule can fire, it might seem that the effect of a non-terminating computation would be to hang the entire system. That this is not the case can be seen by eliminating the synchronous call from the above rule:

$$\begin{array}{l} {*}\text{Lookup}(d,n,a),\ \text{Contents}(d,x) \\ \quad \Rightarrow \ \text{Eval}(\ \text{assoc}(n,x),\ \text{Current},\ p),\ \Sigma(d,n,a,x). \\ {*}p(y),\ {*}\Sigma(d,n,a,x) \ \Rightarrow \ a(y). \end{array}$$

If Eval never returns a result then the second rule will never fire, but this will not prevent other rules from firing. Since all rules involving applicative expressions ultimately reduce to rules in the canonical form and since rules in the canonical form are never non-terminating, we can see that execution can never be stopped by non-terminating applicative expressions.

6.6 Applicative Conditions

Consider the following simulation problem. A relation Sched(x,t) *means that an event x is scheduled to happen at time t* and a predicate Clock(t) means that the current time is t. We want to write a rule to cause some effect E whenever the clock time is at least as late as the time at which an event is scheduled to happen. We allow this rule to be written as follows:

$${*}\text{Sched}(x,t),\ \text{Clock}(t')\ \text{if}\ t' \geq t \ \Rightarrow \ E$$

In general we allow any Boolean-valued applicative expression to appear following an "if" in the cause part of a rule. To illustrate the reduction of these rules to canonical form we show the reduction of the scheduling example:

$\text{Sched}(x,t),\ \text{Clock}(t') \Rightarrow \Pi(t' \geq t),\ \Sigma(x,t,t').$

$*\text{Sched}(x,t),\ \text{Clock}(t'),\ *\Pi(\text{true}),\ *\Sigma(x,t,t') \Rightarrow E.$

$*\Pi(\text{false}),\ *\Sigma(x,t,t') \Rightarrow .$

Here Π and Σ are two new relations associated with this rule. Notice that the relation Sched is not updated until after the value of the applicative expression $t' \geq t$ has been returned by Eval and is known to be true. Also notice that the conditions on Sched and Clock are retested in the second rule. This is because they may no longer be true by the time Eval has returned its result.

As an example of the use of applicative conditions we present a simplified form of the file system described in David Reed's thesis [Reed78]. Let LastRead(r,t) mean that record r was last read at time t and let Value(r,t,x) mean that the value of record r at time t was x. The value of r at any time T can be read and passed to an action A by:

$\text{Value}(r,T,x),\ \text{LastRead}(r,t);\ T \leq t \Rightarrow A(x).$

$\text{Value}(r,T,x),\ *\text{LastRead}(r,t);\ T>t \Rightarrow \text{LastRead}(r,T),\ A(x).$

Updating a record r to a new value y effective at time T is denoted by Update(r,T,y). This operation is only allowed if r has not been read effective at a later time:

$\text{Update}(r,T,y),\ \text{LastRead}(r,t);\ T>t \Rightarrow \text{Value}(r,T,y).$

$\text{Update}(r,T,y),\ \text{LastRead}(r,t);\ T \leq t \Rightarrow \text{AbortUpdate}(T).$

The purpose of AbortUpdate is to clear out any time T updates that might have already been made. It is accomplished by:

```
        AbortUpdate(T), *Value(r,T,x)  ⇒
   else  *AbortUpdate(T)  ⇒  .
```

6.7 State Variables

Often we use private relations to refer to the internal state of an object. For example loc(B,L) might mean that the screen location of a graphic object B is L. It is often convenient to think of these relations as state variables that are private to the object. Consider a rule such as this:

```
Active(self), loc(self,L), *R(x)  ⇒  S(x,L).
```

Clearly L represents the current value of the state variable loc. Therefore we allow this rule to be abbreviated

```
Active(self), *R(x)  ⇒  S( x, @loc).
```

In general, if R is a relation and @R appears in an action of a rule, then we replace @R by a new variable v and place the condition R(self,v) in the cause part of the rule. Notice that this assumes that one or more of the other conditions bind "self." Often this is the "self" bound as a global variable in the cause part of a sequential block.

Figure 1 shows an extended example using all of these abbreviations. It is part of the definition of a graphic object that appears as a square on the screen. The similarity to Smalltalk will be apparent to readers familiar with that language.

```
*Show( self, reply)   ⇒
  { Color[ @scribe, black];
    Draw[self];
    reply(self) }

*Erase( self, reply)   ⇒
  { Color[ @scribe, background];
    Draw[self];
    reply(self) }

*Draw( self, reply)   ⇒
  { Goto[ @scribe, @origin];
    Turn[ @scribe, @tilt];
    DrawAux[ self, 0];
    reply(self) }

*DrawAux( self, 4, reply)   ⇒   reply(self)   else

*DrawAux( self, k, reply)   ⇒
  { Go[ @scribe, @size];
    Turn[ @scribe, 90];
    DrawAux[ @self, k+1];
    reply(self) }
```

Figure 1. Part of a Graphic Object

7. System Structure

In this section we discuss a possible organization for an object-oriented programming system. Although this is not the only possible organization, it will illustrate many of the characteristics of these systems.

Consider the problem of information hiding, that is, making the corresponding relations visible and the non-corresponding relations invisible. For example, in the definition of stacks, the relations NewStack, Push, Pop and Destroy are to be visible to all potential stack users, while the relation Contents is visible only to the implementor of stacks. This is accomplished by placing the names of the corresponding relations in a public directory while keeping "Contents" in the private directory of the implementor. Suppose that all environments contain at least two bindings: "Private" is bound to the private directory and "Public" is bound to the public directory. Then, using the rules for directories defined in Section 6.4 and supposing NewRel[] returns a new relation object, we can define the private relation Contents by:

```
Define[ Private, "Contents", NewRel[] ];
```

The distinction between public and private relations provides a gross level of discrimination between kinds of access. A finer level of discrimination is required to maintain fidelity to the causal model of objects. For example, we might have a class of objects that have a publicly visible attribute Velocity. The value of this attribute might be *determined* by *non-corresponding* causal laws private to this class of objects. Thus it makes sense for other objects to inquire the value of this attribute, but not to alter it. With just the public/private distinction, we only have two choices: (1) make the attribute private, in which case other objects can not use it in conditions, or (2) make the attribute public, in which case it is vulnerable to actions by other objects.

We solve this problem by a simple form of capability based addressing [Dennis66]. We assume that each relation is denoted by a capability, which is a pair <rs,id> in which id is the object identifier for the relation and rs is the set of access rights permitted to possessors of this capability. The three possible access rights correspond to the possible uses of relations in rules: read, add and delete.

When a relation is created a capability bearing all rights is returned. Thus,

```
Define[ Private, "Push", NewRel[] ];
```

defines the private name "Push" to be a new relation that can be used in any way. To restrict access to relations we assume the existence of procedures RemoveR, RemoveA, RemoveD, RemoveRD, ... that create a new capability that is like a given capability except that it has certain rights removed. Thus, if we want to define a public name "Push" with just add-rights that is the same as the private Push, we would write:

```
Define[ Public, "Push", RemoveRD[Push] ];
```

The capability management procedures make use of relations private to the capability manager; these rules have this form:

```
*RemoveR(c,a), Cmap(c,r), Has(c,A), Has(c,D), *AvailCap(d)
   ⇒  Cmap(d,r), Has(d,A), Has(d,D), a(d)
else  *RemoveR(c,a), Cmap(c,r), Has(c,A), *AvailCap(d)
   ⇒  Cmap(d,r), Has(d,A), a(d).
else  *RemoveR(c,a), Cmap(c,r), Has(c,D), *AvailCap(d)
   ⇒  Cmap(d,r), Has(d,D), a(d).
```

Figure 2 shows a complete definition of stacks. In this definition we have used a procedure NewRules[S] which takes a set of rules in symbolic form, S, associates these rules with a new object, activates the rules and returns the object. The resulting rule object is given a private name to allow later operations on it (such as editing or deactivating the rules).

```
Define[ Private, "NewStack", NewRel[] ];
Define[ Private, "Push", NewRel[] ];
Define[ Private, "Pop", NewRel[] ];
Define[ Private, "Destroy", NewRel[] ];
Define[ Private, "Contents", NewRel[] ];

Define[ Private, "Rules", NewRules[
  '*NewStack(a), *Avail(s)  ⇒  Contents(s,nil), a(s).
   *Push(s,x,a), *Contents(s,y)  ⇒  Contents(s,cons(x,y)), a(s).
   *Pop(s,a), *Contents(s,cons(x,y))  ⇒  Contents(s,y), a(x).
   *Destroy(s,a), *Contents(s,x)  ⇒  a(x). ']];

Define[ Public, "NewStack", RemoveRD[NewStack]];
Define[ Public, "Push", RemoveRD[Push]];
Define[ Public, "Pop", RemoveRD[Pop]];
Define[ Public, "Destroy", RemoveRD[Destroy]];
```

Figure 2. Input Commands to Define Stack Objects

Interactive programming is a typical situation in which the object-oriented viewpoint is preferable to the value-oriented viewpoint. Since a person sitting at a terminal responds to conditions and takes actions in time, we consider a person to be an object. To allow people (real objects) to interact with objects in the computer (simulated objects) we represent people by surrogates, called user objects. Therefore users can be put into and removed from relations either by their own actions or by the actions of other objects acting on the users' user objects.

The user object is atomic as far as the users are concerned. At a lower level of abstraction (the system level) the user object resolves into a number of smaller objects representing the display screen, the keyboard, the directories etc.

In their interactions with the rest of the system user objects act like any other objects. However, because these objects act as proxies for people they have some special characteristics. Since the future action of users cannot be defined by a finite set of rules, the system must provide a way of interacting with users. This includes methods of informing them of the relations that hold on their user objects and means for allowing them to direct their user objects to take actions. Both of these can be accomplished by allowing users to enter rules or parts of rules.

To account for the fact that users might never repeat their actions under the same conditions, the commands that users type are formally considered to be part of an open-ended sequential block. Thus each command is implicitly parameterized by an ever-changing attribute that can be thought of as time. The definitions shown in Figure 2 are typical commands that a user might type at a terminal.

All the commands in Figure 2 are synchronous calls - the action must be completed (or aborted) before the user can continue. Parallel activities can be initiated by simple unconditional actions, for example:

```
Compile( Progl, Replyl), Compile( Prog2, Reply2);
```

The user can also type actionless conditions, e.g.,

```
                    Replyl(result)  ⇒  ;
```

which causes the user task to pause until the conditions are satisfied. Complete rules are useful for recovering results from parallel tasks, e.g.,

```
    *Replyl(result)  ⇒  Define[ Private, "Binary", result];
```

Since applicative expressions are allowed as parts of actions users can also call for the evaluation of applicative expressions:

```
     Display[ fac(3) ];
        6
```

Thus the command language is the same as the object-oriented language; the value-oriented language is embedded in the object-oriented language. This seems to be the best relationship of object- and value-oriented languages; it can be seen in several existing systems (e.g., LISP and Backus's AST).

8. Formal Semantics

8.1 Abstract Syntax

In this section we present a formal semantics for the execution of condition-action rules. (The casual reader is advised to skip this section.) The formal semantics will be expressed as a series of function definitions that define a mapping from abstract rules into state transition functions. The abstract syntax for rules is shown in Figure 3; the correspondence between the abstract and concrete syntaxes should be obvious.

rule	=	cause effect
cause	=	predicate$^+$
effect	=	predicate$^+$
predicate	=	name constructor + ~ name constructor
constructor	=	item*
item	=	name + constant + constructor

Figure 3. Abstract Syntax for Rules

8.2 State Space

As discussed in Section 2 (Intension versus Extension) intensionally different relations are distinguished by having different IDs. Since the extensions of intensional relations can change in time, a state is considered a mapping from IDs into extensional relations. The definition of the state space is shown in Figure 4.

states	=	IDs $\rightarrow$ relations
relations	=	P(tuples)
tuples	=	elements*
elements	=	objects + values + tuples
objects	=	atomic-objects + relation-objects

where P(s) denotes the powerset of the set s.

Figure 4. The State Space

Recall that relations are manipulated through <u>capabilities</u> that control read, add and delete access to the underlying relations. Therefore we define the set relation-objects to be the set of all capabilities. This set has three exhaustive but not mutually exclusive subsets,

$$\text{ReadCaps, AddCaps, DeleteCaps} \subseteq \text{relation-objects}$$

$$\text{relation-objects} = \text{ReadCaps} \cup \text{AddCaps} \cup \text{DeleteCaps}$$

Note that a capability that bears multiple rights (e.g., add and delete) will be a member of several of these sets (e.g,, AddCaps and DeleteCaps). The function Cmap takes a capability to its underlying relation ID:

$$\text{Cmap: relation-objects} \rightarrow \text{IDs}$$

These notions are formally defined in Figure 5.

rights	=	$\{R, A, D\}$
rights-sets	=	P(rights)
relation-objects	=	rights-sets $\times$ IDs
Cmap(r)	=	$\underline{2}(r)$
ReadCaps	=	$\{c \in \text{relation-objects} \mid R \in \underline{1}(c)\}$
AddCaps	=	$\{c \in \text{relation-objects} \mid A \in \underline{1}(c)\}$
DeleteCaps	=	$\{c \in \text{relation-objects} \mid D \in \underline{1}(c)\}$

Figure 5. Capabilities

8.3 Multiple-valued Functions

Our goal in this section is to define a state transition function for condition-action rules. Notice, however, that the execution of rules is in general non-deterministic. That is, from the current state the state transition function may define several successor states. In other words, the state transition function is multiple valued. What this means of course is that we have a state transition relation rather than a state transition function. Although we will define a relation Cycle $\subseteq$ states $\times$ states such that Cycle(s,s') is true if and only if s' is a possible successor state of s, it will often be more convenient to think of Cycle(s) as a multiple-valued function. This will allow us to use a more familiar functional notation in our definitions. Unless specified otherwise, in the following subsections "function" will refer to both single-valued and multiple-valued functions.

Since we will be dealing with multiple-valued functions, it will be necessary to talk about sets of multiple-valued functions. We write $D \Rightarrow R$ for the set of all multiple-valued functions with domain D and range R. This notation is analogous to $D \rightarrow R$, the set of all single-valued functions from D into R. Of course mathematically $D \Rightarrow R$ is just $D \times R$, but our notation will better emphasize the functional viewpoint.

Although we introduce additional notation as we need it, the reader can find a complete description of the functional and relational operators we use in [MacLennan81] and [MacLennan83].

8.4 Semantics of Rules

Our goal in the following sections is to define a function

$$\text{Cycle: states} \Rightarrow \text{states}$$

that describes the non-deterministic transition from state to state. We will do this by first defining a single-valued function

$$\text{Rule: rule} \rightarrow \text{envs} \times \text{states} \Rightarrow \text{states}$$

that takes rules into multiple-valued functions that take environment-state pairs into states. Note that the arrows are right associative and less binding than the cross. We use the convention of writing the names of abstract syntactic categories in lowercase (e.g., "rule") and the names of the corresponding semantic functions with a leading uppercase letter (e.g., "Rule").

We will define the Rule function in a top-down order. The effect of the cause part of a rule is to test for the cause being true of the state and, if it is, to extend the environment by the names bound in the pattern matching process. This extended environment is then used during execution of the effect part of the rule. The definition of Rule is:

```
Rule[c,e](E,s)  =  Effect[e] E' s
  where E'  =  Cause[c] s E
```

where c and e are the cause and effect parts of the rule and E' is the extended environment. Note that function application is assumed to be left associative; thus Fxy means (Fx)y.

We first address the effect part of a rule because it is a little simpler. The type of Effect is:

$$\text{Effect: effect} \rightarrow \text{envs} \rightarrow \text{states} \Rightarrow \text{states}$$

The effect part of a rule is composed of a series of actions that are executed in order in the given environment. Thus the state transition function defined by the effect is just the composition of the state transition functions defined by its constituent actions:

$$\text{Effect}[A_1, A_2, \ldots, A_n]\,E$$
$$= (\text{Action}[A_n]E) \cdot \ldots \cdot (\text{Action}[A_2]E) \cdot (\text{Action}[A_1]E)$$

There are two kinds of actions: additions and deletions. The type of their semantic function is

$$\text{Action: predicate} \rightarrow \text{envs} \rightarrow \text{states} \Rightarrow \text{states}$$

To process an addition, nc, whose name part is n and whose constructor part is c, we must evaluate c in the current environment and add it to the relation that results from looking up n in the current environment. The updated relation is

$$[s \cdot \text{Cmap} \cdot E]n \cup \{\text{Constructor}[c]E\}$$

where s is the current state, Cmap is the capability mapping function and E is the current environment. This new relation must replace the old value associated with the relation ID in the state. We use [x:y]/f to mean a function like f except that x is mapped to y. Therefore the new state is defined by

$$\text{Action}[nc]Es =$$
$$[([\text{AddCaps} \rightarrow \text{Cmap}](En)) : ([s \cdot \text{Cmap} \cdot E]n \cup \{\text{Constructor}[c]E\})]/s$$

Here AddCaps $\rightarrow$ Cmap is the capability mapping function with its domain restricted to add-capabilities, since $s \rightarrow f$ means the function f with its domain restricted to s. This ensures that we do not add to a relation unless we have an add capability for it.

The semantics for deletions is analogous:

$$\text{Action}[-nc]Es =$$
$$[(([\text{DeleteCaps} \rightarrow \text{Cmap}](En)) : ([s \cdot \text{Cmap} \cdot E]n - \{\text{Constructor}[c]E\})]/s$$

The formal semantics for constructors is straight-forward:

$$\text{Constructor: constructor} \rightarrow \text{envs} \rightarrow \text{tuples}$$
$$\text{Constructor}[\text{item}_1 \ldots \text{item}_n]E$$
$$= \langle \text{Item}[\text{item}_1]E, \ldots, \text{Item}[\text{item}_n]E \rangle$$

$$\text{Item: item} \rightarrow \text{elements}$$
$$\text{Item}[\text{name}]E = E(\text{name})$$
$$\text{Item}[\text{constant}]E = \text{constant}$$
$$\text{Item}[\text{constructor}] = \text{Constructor}[\text{constructor}]$$

Next we address the cause part of rules. The type of the semantic function for causes is:

$$\text{Cause: cause} \rightarrow \text{states} \rightarrow \text{envs} \Rightarrow \text{envs}$$

As expected, cause parts leave the state unchanged but (temporarily) modify the environment. The semantics of a cause is just the composition of the semantics of its constituent conditions:

$$\mathit{Cause}[C_1, C_2, \ldots, C_n]s$$
$$= (\text{Cond}[C_n]s) \cdot \ldots \cdot (\text{Cond}[C_2]s) \cdot (\text{Cond}[C_1]s)$$

The semantics of conditions is defined by the function Cond:

$$\text{Cond: condition} \rightarrow \text{states} \rightarrow \text{envs} \Rightarrow \text{envs}$$

First we address the semantics of positive conditions, Cond[nc]sE, where n is the name of a relation and c is a constructor. To accomplish this we define a function

$$\text{match: constructor} \rightarrow \text{relations} \rightarrow \text{envs} \Rightarrow \text{envs}$$

that will determine if the pattern c occurs in the relation that is the meaning of n in the current environment and state. Thus,

$$\text{Cond[nc]sE} = \text{match[c](ReadRel s E n)E}$$

where (ReadRel s E n) is a (readable) relation whose name is n in the environment E and the state s:

$$\text{ReadRel s E} = \text{s}\cdot[\text{ReadCaps} \rightarrow \text{Cmap}]\cdot\text{E}$$

Next consider match[c]RE; this must determine if the pattern c occurs in the relation R and, if it does, return an environment that is an extension of E that includes the bindings made by the pattern matching process. Now, suppose we have a function FindEnvs:

$$\text{FindEnvs: relations} \rightarrow \text{envs} \rightarrow \text{P(envs)}$$

such that FindEnvs[c]RE returns all extensions of E that can result from matching c in R. We want match to be a multiple-valued function that can return any one of these extensions of E. Therefore, define

$$\text{match} = \in^{-1}\cdot\text{FindEnvs}$$

where $\in(x)$ is any set containing x so $\in^{-1}(S)$ is a multiple-valued function that returns any element of the set S.

Consider the function Constructor[c]E; this evaluates the constructor c in the environment E to yield a tuple t. Therefore (Constructor[c]) $^{-1}$t is any environment in which the evaluation of c yields t. Therefore (Constructor[c]) $^{-1}$ matches the pattern c against a tuple and yields any environment that allows the match to succeed. Its type is

$$(\text{Constructor}[c])^{-1}\text{: tuples} \Rightarrow \text{envs}$$

If unimg[f]x represents the set of all y such that f(x,y) then unimg[(Constructor[c]) $^{-1}$]t is the set of all such environments and its type is

$$\text{unimg}[(\text{Constructor}[c])^{-1}]\text{: tuples} \Rightarrow P(\text{envs})$$

Now suppose that we have a function (defined later)

$$\text{minext: envs} \rightarrow P(\text{envs}) \Rightarrow P(\text{envs})$$

such that minext E S is the set of all the minimum extensions of E that are in S. It is then easy to see that

$$[\in^{-1} \cdot (\mathit{minext}\ E) \cdot \mathit{unimg}[(\mathit{Constructor}[c])^{-1}]]t$$

is any minimum extension to E that results from matching pattern c against tuple t. Now, if img[f]S is the set of all y such that for some x $\in$ S we have f(x,y), then it is easy to see that

$$\text{img}[\in^{-1} \cdot (\text{minext } E) \cdot \text{unimg}[(\text{Constructor}[c])^{-1}]]R$$

is the set of all extensions to E that can result from matching c against any tuple in R. Simplifying this expression we get our definition for FindEnvs:

$$\text{FindEnvs}[c]RE = [(\text{minext } E) \cdot \text{img}[(\text{Constructor}[c])^{-1}]]R$$

It remains to define minext E S, the subset of S containing the minimum extensions of E. Observe that unimg[$\subseteq$]E is the set of all supersets (extensions) of E. Therefore

$$S \cap \text{unimg}[\subseteq]E$$

is the subset of S containing just extensions to E. We want to find the minimum elements of this set, which is just the initial members of the subset relation restricted to this set:

$$\text{minext } E\ S = \text{init}[\ \subseteq \uparrow (S \cap \text{unimg}[\subseteq]E)\ ]$$

This completes the definition of positive conditions.

The semantics of negative conditions is simpler than positive conditions. First define

$$\text{Cond}[\neg nc]sE = \text{nomatch}[c](\text{ReadRel } s\ E\ n)E$$

Negative conditions are simpler because they don't update the environment. Therefore, nomatch[c]R is just an identity function restricted to those environments in which c doesn't match a tuple in R. Recall that FindEnvs[c]RE is the set of all extensions to E such that c matches a tuple in R. Therefore, if FindEnvs[c]RE = $\emptyset$ then there are no such environments; in other words c does not match any tuples in R. Thus, the set of all environments in which c doesn't match a tuple in R is the inverse image of the empty set under FindEnvs[c]R. By using this set to restrict the domain of the identity function we get the definition of nomatch:

$$\text{nomatch}[c]R = [\text{unimg}\ (\text{FindEnvs}[c]R)^{-1}\ \emptyset] \rightarrow \text{Id}$$

where Id is the identity function.

8.5 Semantics of Execution

We have now described the semantics of an individual rule; it remains to define the state transition semantics of the entire system. Recall that a rule has to be evaluated in the proper environment. Therefore we postulate the existence of an ID p ∈ IDs such that sp(i,r,E) means that i is the ID of a rule r that must be evaluated in environment E. Hence,

$$\text{sp}: \ \text{Ids} \rightarrow \text{rule} \times \text{envs}$$

Next we define a function Trans i s, which means a transition of state s by rule object i:

$$\text{Trans}: \ \text{IDs} \rightarrow \text{states} \Rightarrow \text{states}$$
$$\text{Trans } i\ s = \text{Rule}[r](E,s) \quad \text{where} \quad (r,E) = \text{sp}\,i$$

Thus Trans(i) is the state transition function for rule object i.

To complete the state transition semantics we need a multiple-valued function Cycle such that Cycle(s) is any state that follows immediately from s. Thus,

Cycle: states $\Rightarrow$ states

We want Cycle(s,s') to be true if there is any rule object i such that Trans[i](s,s'). Now, Rng Trans is the range of Trans, i.e., the set of all Trans[i] for any i. Therefore $\bigcup$(Rng Trans) combines all of these individual transition functions into one. Cycle can now be defined:

$$\text{Cycle} = \bigcup(\text{Rng Trans})$$

Notice that Cycle(s,s') means that state s can lead to state s' in one transition; we would like to extend this to repeated transitions. Notice however that object-oriented systems are often non-terminating and non-deterministic, therefore their denotational semantics can not take the form of a single-valued function. Rather, a denotational semantics for an object-oriented system is a relation that relates past states to possible future states. Defining this relation is our next task.

Let Cycle* be the reflexive transitive closure of the relation Cycle; then Cycle*(s,s') means that state s can lead to state s' in zero or more transitions. Thus we define CanReach = Cycle* so that CanReach(s,s') means that state s can lead to state s'.

The definition of CanReach is too permissive for many purposes since CanReach(s,s') is true if there is any set of intermediate states that will get us from s to s'. Sometimes we want to know if a state can lead to another state that is a dead end, i.e., that we can't get out of. We can express this idea as follows. Let const($\perp$) represent a constant function that always returns a distinguished value, $\perp$. Then extend Cycle to a total function

$$\text{Cycle} \,/\, \text{const}(\perp)$$

so that [Cycle/const($\perp$)]s is $\perp$ if Cycle(s) is undefined. Notice also that this total function is $\perp$ preserving. Now consider the relation

$$\text{Reaches} = [\text{Cycle} \,/\, \text{const}(\perp)]^*$$

If Reaches(s,$\perp$) then we know that s can lead to an state that we can't get out of. If this is not true, then Reaches(s,s') means that s can safely reach s'.

9. Summary

Several similarities will have been observed between this work and previous work. For example, in [Lomet76] and [Lomet80] David Lomet describes a distinction between values and objects that is very similar to ours. We recommend these papers to the reader as an interesting alternate approach to the value/object distinction. However, we believe that our simulation based notion of an object is more fundamental than Lomet's storage based idea.

As has been noted, the idea of a set of alterable relations on a finite universe of objects is similar to the knowledge representation networks used in artificial intelligence applications and the set of causal laws has similarities to both production systems and PROLOG programs. A fundamental difference between our work and these systems is that our notion of a production does not require any backtracking. Furthermore, we know of no other attempt to make these ideas the basis of an entire programming system.

This paper extends previous work in several areas. First, it clarifies the nature, purpose, and roles of values and objects in programming languages. Second, it argues that objects are important programming devices and should be included in applicative languages. Finally, it proposes a specific form in which objects can be accommodated. This includes the distinction between corresponding and non-corresponding objects and relations, and the use of name contexts and capabilities to hide non-corresponding objects and relations. Finally, it proposes that the manipulation of objects be specified by causal laws that determine sets of transactions to alter the state. It is shown how these concepts may be combined with applicative programming ideas.

10. Acknowledgement

Research leading to and the preparation of this report was supported by the Office of Naval Research under contract number N00014-82-WR-20162 and the Foundation Research Program of the Naval Postgraduate School with funds provided by the Chief of

Naval Research.

11. References

[Dahl70] Dahl, O.-J., Myhrhaug, B. and Nygaard, K., _The SIMULA 67 Common Base Definition_, Publication S-22, Norwegian Computing Centre, Oslo, 1970.

[Dennis66] Dennis, Jack B. and Van Horn, Earl C., Programming semantics for multiprogrammed computations, _CACM 9_, 3 (March 1966), pp 143-155.

[Kay77] Kay, Alan C., Microelectronics and the personal computer, _Scientific American 237_, 3 (September 1977), pp 230-244.

[Lomet76] Lomet, David B., Objects and values: the basis of a storage model for procedural languages, _IBM Journal Res. and Dev. 20_, 2 (March 1976), pp 157-167.

[Lomet80] Lomet, David B. A data definition facility based on a value-oriented storage model, _IBM Journal Res. and Dev. 24_, 6 (November 1980), pp 764-782.

[MacLennan73] MacLennan, B. J., Fen - an axiomatic basis for program semantics, _CACM 16_, 8 (August 1973), pp 468-471.

[MacLennan75] MacLennan, B. J., _Semantic and Syntactic Specification and Extension of Languages_, PhD Dissertation, Purdue University, 1975.

[MacLennan81] MacLennan, B. J., Introduction to relational programming, Proc. ACM Conf. Functional Prog. Lang. and Comp. Arch., October 18-22, 1981.

[MacLennan82] MacLennan, B. J., Values and objects in programming languages, SIGPLAN Notices 17, 12 (December 1982), pp 70-79.

[MacLennan83] MacLennan, B. J., Overview of relational programming, SIGPLAN Notices 18, to appear (1983), see also Naval Postgraduate School Computer Science Department Technical Report NPS52-81-017 (November 1981).

[Maisel72] Maisel, H. and Gnugnoli, G., Simulation of Discrete Stochastic Systems, SRA, 1972.

[Pritsker79] Pritsker, A. Alan B. and Pegden, Claude Dennis, Introduction to Simulation and SLAM, John Wiley, 1979.

[Reed78] Reed, David P., Naming and Synchronization in a Decentralized Computer System, PhD Dissertation, 1978, MIT/LCS/TR-205.

[Rentsch82] Rentsch, Tim, Object oriented programming, SIGPLAN Notices 17, 9 (September 1982), pp 51-57.

[Robson81] Robson, David, Object-oriented software systems, BYTE 6, 8 (August 1981), pp 74-86.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center 2
Cameron Station
Alexandria, VA 22314

Dudley Knox Library 2
Code 0142
Naval Postgraduate School
Monterey, CA 93940

Office of Research Administration 1
Code 012A
Naval Postgraduate School
Monterey, CA 93940

Chairman, Code 52Hq 40
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93940

Professor Bruce J. MacLennan, Code 52Ml 12
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93940

Dr. Robert Grafton 1
Code 433
Office of Naval Research
800 N. Quincy
Arlington, VA 22217

Dr. David W. Mizell 1
Office of Naval Research
1030 East Green Street
Pasadena, CA 91106

John M. Hosack 1
Department of Mathematics
Colby College
Waterville, ME 04901

Dr. David B. Lomet 1
IBM Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Jim Bowery 1
Viewdata Corporation
3rd Floor
1111 Lincoln Road
Miami Beach, FL 33139

J. Craig Cleaveland 1F35 Bell Laboratories 1600 Osgood Street North Andover, MA 01845	1
Professor John M. Wozencraft, 62Wz Department of Electrical Engineering Naval Postgraduate School Monterey, CA 93940	1
Mark Himmelstein 1323 Tulip Way Livermore, CA 94550	1

4 -

DT